F2C - A F77 TO C CONVERTER

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

J. R. MAHESH KUMAR

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

AUGUST, 1991

ACKNOWLEDGEMENTS

I am extremely indebted to Dr. Sanjeev Kumar - my thesis supervisor, for the valuable guidance he gave me at every stage of the preparation of this thesis. Working with him, in the very cordial and yet professional atmosphere he created, has been a special and memorable experience for me.

Everyone who has prepared a thesis report knows about, the innumerable irksome details one has to take care of, to bring a document to the camera ready form. Without the help of Raashi and Raaghu - who prepared figures, formatted the text and did these things all over again as I made changes - document preparation would have been a painful experience. I am highly obliged to them for their help.

Thanks are due to Shankar and Kiran for honing my FORTRAN77 knowledge and providing test programs.

My thanks to all the members of the faculty of the Department of Computer Science and Engineering IITK, who, with their able guidance, made my pursuit of M.Tech. here, an intellectually rewarding exercise. Thanks also to the nonteaching staff of the Department who helped in various ways.

My special thanks to all the faculty and student friends of Kannada Sangha who made life at IITK a very pleasant experience - an experience I would cherish always.

CERTIFICATE

It is certified that the work contained in the thesis entitled F2C - A F77 to C Converter, by J R Mahesh Kumar has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

(Sanjeev Kumar)
Assistant Professor
Dept. of Computer
Science and Engg.
I I T Kanpur

ABSTRACT

F2C is a source code converter which translates programs in ANSI standard FORTRAN77 to equivalent programs in the C language. The converter is built using general purpose language processing tools which take high level specifications and generate programs which can be combined to produce the converter. Programming in a traditional programming language has been kept to a minimum.

CONTENTS

LIST OF FIGURES

1. TO BEGIN WITH ..

Source code converters convert programs written in one programming language to equivalent programs in another programming language.

Why should one think of converting programs from one language to another?

We see that in the past thirty years, there has been an explosive growth in the use of computers and a large amount of software has been developed, tested and put into commercial use. This software has been developed on many different hardware platforms using widely differing operating system environments and using diverse programming languages. With the growth in technology, there arise increased performance expectations and moving over to newer and better hardware systems becomes inevitable. But what is to be done with the existing programs - developed on older systems and already thoroughly tested and proven? [1]

One would hope that these programs would run equally well on the newer systems too. This would be the expectation at least for those programs written in well known and widely used high level languages like, FORTRAN or COBOL .

But unfortunately, even the ´standard´ languages supported by most of the vendors come with each vendor providing his own nonstandard ´extensions´ and ´enhancements´ and ´improvements´ resulting in a proliferation of widely differing dialects of the same language. Over fifty dialects of COBOL are said to be extant! At least a dozen versions of FORTRAN exist!

Rewriting the programs entirely in the new dialect becomes prohibitively costly. So programs are ported to new dialects and / or environments. It would not be an exaggeration to say that a significant portion of software development activity in the industry involves 'fixing' old software onto new systems.

Manual translation of code from one dialect to another has many undesirable aspects to it:

1. The costs associated with software lbr.

2. The costs related to testing every single program so translated.

3. The loss due to 'down time' of software during translation.

The reasons mentioned above argue definitively for automating source code conversion. Automatic converters once tested, eliminate the need to test the programs translated by them.

Many automatic converters for translating one dialect of COBOL to another dialect of the same language have been developed in the industry.

1.1 Different dialects ... Fine. But different languages?

There are other situations very different from those described above, where conversion of code from one language to - not a different dialect of the same language - but a completely different language becomes very desirable.

Consider the Ada Language sponsored by the Defense Department of the United States of America. The Defense Department, which also happens to be one of the biggest software customers in U.S., insists that all software developed for it be written in the Ada Language. This has naturally led to a widespread usage of Ada and now extensive language support tools such as debuggers etc., are available for Ada. Because of this, not only the Defense Department but also other customers who have Ada platforms prefer Ada to other languages. So those companies which have developed applications in other languages now want to have their programs converted into Ada programs. Presently, many automatic converters to convert programs from other major programming languages to Ada have entered the U.S. market.

Consider another situation where, a FORTRAN programmer who wants to shift over to using the C language for writing programs so that he can make use of the better control constructs and data structure supports it provides. If he has already written many applications using FORTRAN he would not want to dump them. And he would continue to use FORTRAN and build more applications. This would prevent him from ever shifting over to C. A FORTRAN to C converter could help make this transition smoothly.

1.2 My Thesis

My thesis involves developing a converter from ANSI standard FORTRAN-77 [2] to the C [3] language (F2C).

With the rapid spread of the UNIX operating system in the last ten years it has become the *de facto* industry standard for operating system environments. Programmers have found the UNIX programming environment extremely helpful in developing programs. UNIX has been written in C and perhaps because of that, the entire operating system environment has the C-philosophy, C-thinking ingrained into it. To make use of UNIX and the innumerable programming tools built on it in an effective and efficient way learning C is almost indispensable. A programmer using any other language would find himself handicapped in integrating his programs with the tools available on the system to build useful applications quickly. No other language is as well supported as C and no other language blends into the environment so smoothly. Hence a large number of UNIX users who programmed in other languages earlier want to shift to writing programs in C now.

For a FORTRAN programmer, switching over to C is possible if and only if his existing FORTRAN programs can be saved for his use with the new programs he would be writing in C. The object codes of the two languages are not compatible because of differences in parameter passing mechanisms and data storing methods. Further, large quantities of standard software - especially, Mathematical Software Libraries, - written in FORTRAN and compatible only with FORTRAN programs - have been available to a FORTRAN programmer for a long time now. The application areas of a traditional FORTRAN programmer have been such that without the support of these Mathematical Software Libraries, little useful programming can be done. Software support in these areas are not extensively available currently to a C programmer because of the different needs of a traditional C programmer. It would be necessary to build these Mathematical Software Libraries in C if switching over to C usage from FORTRAN is to be possible. Rewriting these huge libraries in C would entail enormous software development effort.

With these factors considered, the need for an automatic source code converter from FORTRAN to C would be very obvious.

Of the several versions of FORTRAN currently in use in the industry, I have chosen the ANSI standard FORTRAN77 as the source language to be converted because :

1. This standard has been in wide use for more than ten years now and all the FORTRAN compilers marketed in the last ten years support this standard.

2. The differences if any between the different compilers are in those features not fixed by the ANSI standard, and these differences are small

enough so that the converter needs to be enhanced only slightly to accommodate these extensions to the standards.

1.3 Doing it with Tools

My thesis is not limited to just developing a source code converter. It incorporates another important idea, viz., to generate the converter from high level specifications of the source and target languages using tools for program generation.

While it is certainly possible to write the entire converter program in a traditional programming language like C, the development costs would be enormous. To move from the prototype to the working version, to change specifications as the program develops, all these become very costly in terms of time and effort. Using program generating tools to build the converter helps not only to cut down these costs but also to concentrate on the design aspects better, so that we will be able to produce a converter that is likely to be more reliable.

So the philosophy is - work with only specifications, leave it to the tools to generate programs.

2. THE DESIGN

2.1 Source Language Converters

Language converters are programs very similar to compilers. For a compiler, the source language is a high level language and the target language is a machine language. In the case of a converter both the source and the target are likely to be high level languages. This means that the front end of both compilers and converters will be very similar. Both involve the analysis of the source language program and representing the source program in some form of intermediate representation suitable for further processing.

Therefore, we will be employing methods and program generating tools used in compiler construction for lexical and syntax analyses of the source language.

The use of tools for generating lexical analysers, and parsers from high level specifications for these programs is well understood and routinely employed in the industry. However, the use of such tools for syntax tree construction, and further processing of these syntax trees is not as common as it should be. In the development of F2C an attempt has been made to minimise traditional programming and use language processing tools at all stages to generate converter from language specifications.

2.2 The Structure of a Converter

The organisation of a converter is as shown in the figure 2.1. It can be divided into five stages:

1. Lexical Analysis: The source program is read by the lexical analyser which produces lexical units(tokens) received by the next stage.

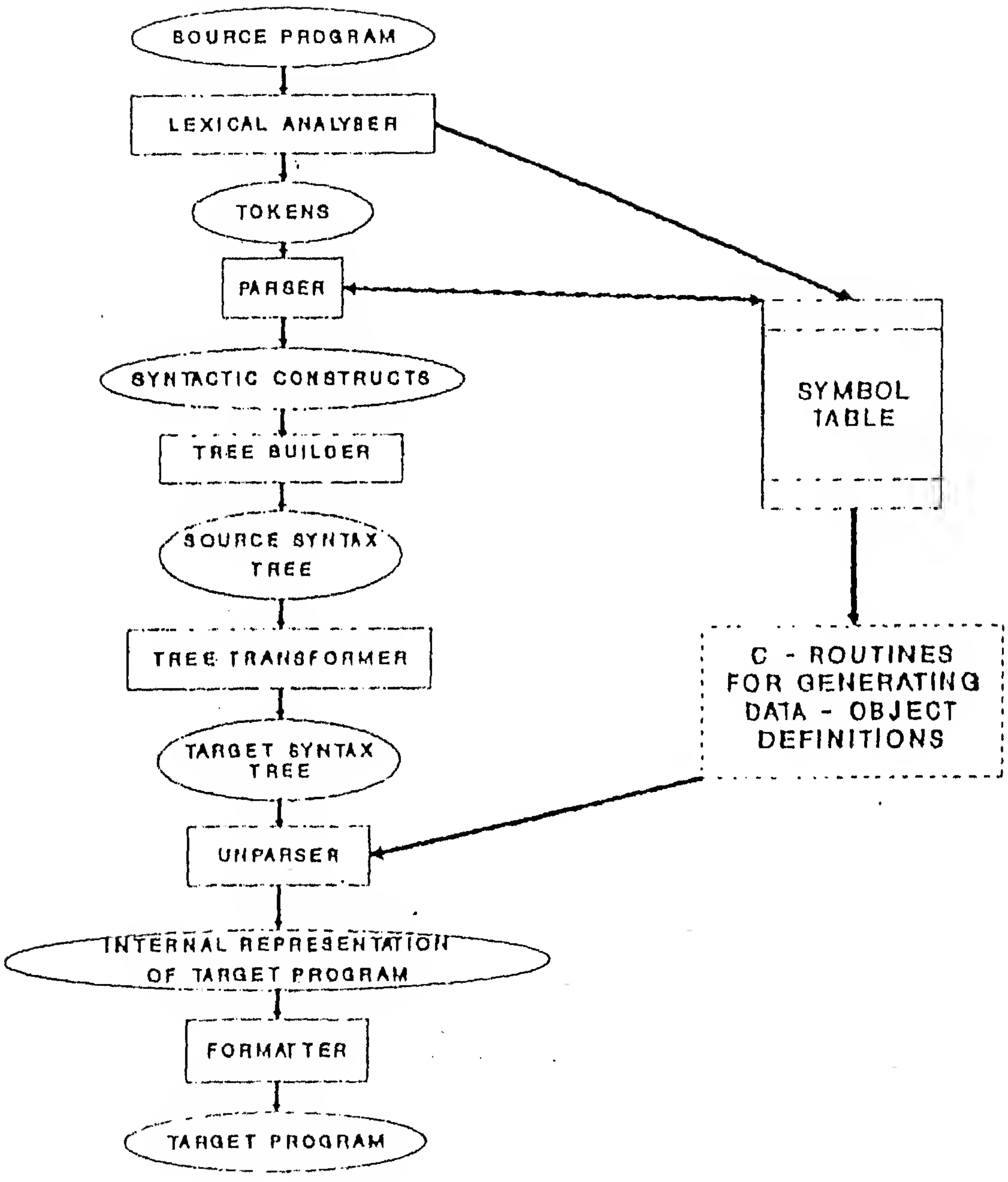

Figure 2.1 Structure of a Source to Source Program Converter

2. Syntax Analysis: The stream of tokens produced by the lexical analyser is received by the parser which recognises the different syntactical constructs. At this stage we also do some semantic checking which mainly involves collecting various attributes of different identifiers and storing them in symbol table.

3. Tree Building: This stage involves building syntax trees for the source program as and when the different lexical constructs are recognised by the parser. So at the end of this stage the syntax tree built, together with the symbol table contains all the information about the input program necessary for making language conversion.

4. Tree Transformation: At this stage the syntax tree built by the tree builder is transformed into the syntax tree of the target language. This is achieved by applying various transformations on different parts of the original syntax tree. It is at this stage that the actual conversion of source program to semantically equivalent target program takes place.

5. Unparsing the Syntax Tree: During this stage the converter unparses the transformed syntax tree now representing the original program in target language syntax. The information in the syntax tree along with that in the symbol tables are used at this stage to generate the target program in some form of internal representation.

6. Formatting: The output of the unparser and other routines will be in a coded form containing formatting details and other information. The formatter converts this into the actual target program.

.3 The Tool Kit

Many different language processing tools have been used or developing F2C. These are general purpose tools suited for various language processing applications.

Lex [4] - The Lexical Analyser generator: This is a well known tool on Unix. It takes regular expression specifications for tokens and produces a lexical analyser capable of recognising these tokens.

Yacc [5] - The Parser generator: This tool takes LALR(1) grammar specifications and produces a parser capable of recognising inputs satisfying the grammar specifications. This requires a routine to act as a lexical analyser. Such a routine is generally built using Lex and hence Yacc has been designed to work well in conjunction with Lex.

Treegen [6] - The Tree Manager: This tool is a new addition to the kit of a language processor builder. This tool is useful in all those situations where it is convenient to store information in the form of a tree and process this information by applying suitable transformations on the tree structure. The processed information can be retrieved later from the transformed tree. This tool has been used in F2C for building syntax trees, transforming them and for unparsing them.

2.4 More about Treegen

Since Treegen is a relatively new language processing tool, it is perhaps necessary here to give a more detailed description of the tool.

Treegen is a tool which from a set of high level input specifications, generates routines to perform four different but related sets of tasks.

1. Tree builder: These are routines which help build a tree according to the NODE specifications given.

2. Tree Transformer: These routines help transform different portions of the tree built according to a set of RULEs given in the input specifications.

3. Tree Unparser: These routines help to unparse the tree according to a set of unparse specifications. They also provide means to execute user defined functions at user specified points in the course of unparsing the tree.

4. Formatter: The output of the unparser is in a coded form with formatting details and other information. This needs to be converted into user readable form. This is done by the formatter. This formatter is not actually generated by Treegen. However it is a program developed to work exclusively with Treegen.

5. Symbol Table Handler: These routines can be used in language processing applications to build symbol tables, resolve references to identifiers taking into consideration different kinds of scope rules.

   This part of the Treegen tool has not been used in building F2C because the methods for collecting attributes of symbols and resolving references to symbols in FORTRAN77 are very different from those of most other programming languages and Treegen does not have provisions to support the rather unusual needs of FORTRAN77.

2.5 The Structure of F2C

The Figure 2.2 shows the complete structure of F2C with data flow information, the different tools used for generating various components of F2C and the type of input specifications

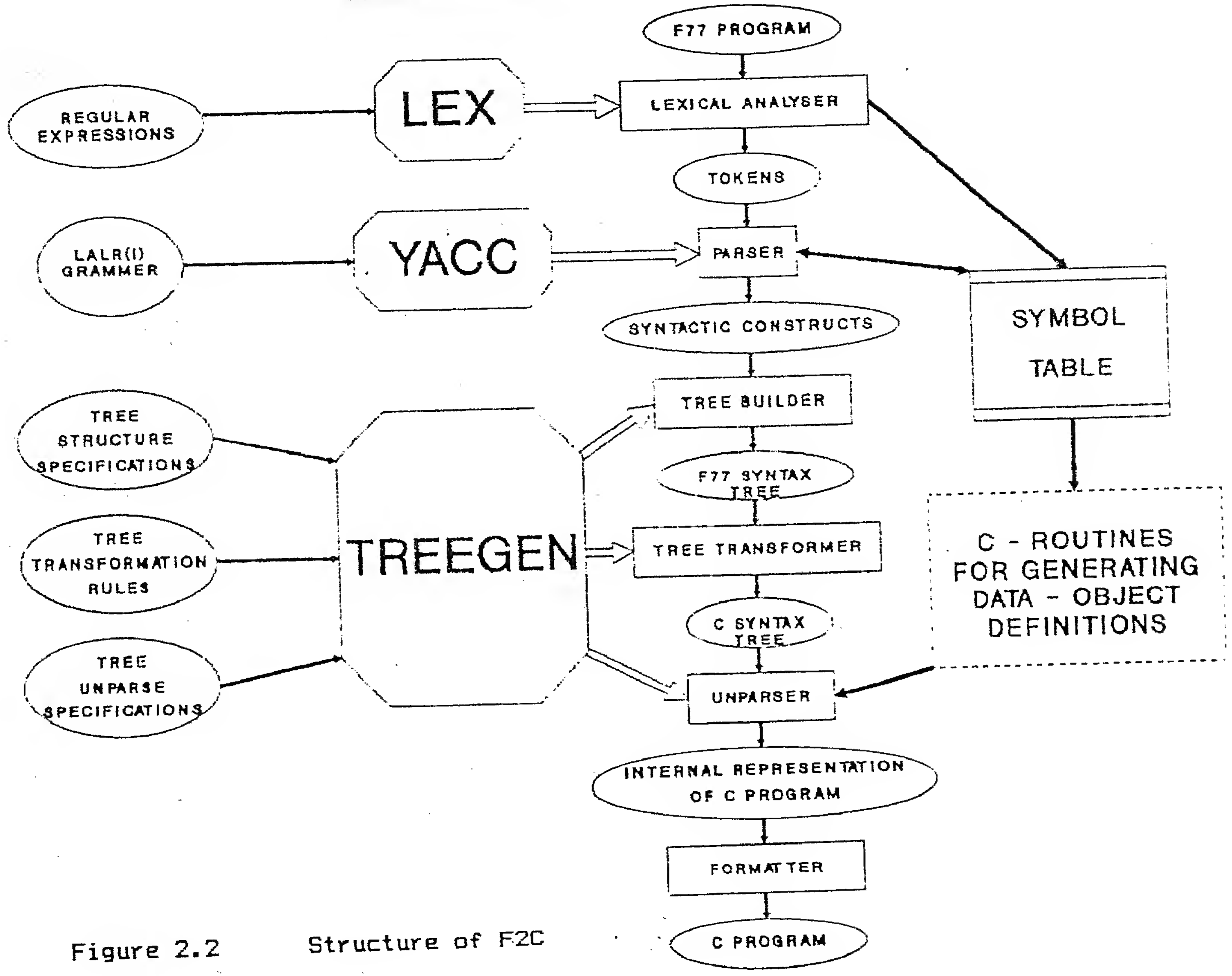

Figure 2.2 Structure of F2C

needed by these tools. It can be seen that except the routines needed for symbol table management and those needed for generating data object definitions from these symbol tables at the time of tree unparsing all other parts of the converter are generated with the help of the tools. This method has been one of the main features of F2C.

3. FROM IDEAS TO PROGRAMS

In this chapter we shall consider some of the issues nvolved in implementing a FORTRAN77(F77) to C converter.

.1 Lexical and Syntax analysis

The lexical and syntax analysis of FORTRAN (of all hues) s more difficult than those of many other languages because of ome unusual features of FORTRAN. These problems are well known nd do not require elaboration here.

In brief, the lexical analysis is complicated by these 'acts:

1. Blanks, tabs and newlines(whitespace) can be embedded inside a single lexical unit(token).

2. There may not always be a field separating character between two different tokens.

3. Keywords are not reserved. One may use them as identifiers.

Syntax analysis is made difficult because :

1. Array element reference and function reference have identical syntax.

2. The Statement Function Statement which is a component of the Specification Part has a syntax which could be that of an Assignment Statement occurring in the Execution Part.

A problem specific to a converter like F2C and not encountered by a compiler writer is this:

> Comments are to be preserved in the converted code and hence must be accommodated in the grammar. But comments can occur virtually anywhere in the program - even within a token - because of continuation lines.

3.2 A Typical F77 program

The stages beyond parsing in the building of a converter from F77 to C may be explained best with the help of an actual F77 program.

```
 1   C      FUNCTION TO COMPUTE THE VECTOR SUM OF A 1-D ARRAY
 2   C      OF INTEGERS AND FIND IF IT IS ZERO.  RETURNS .TRUE.
 3   C      IF VECTOR SUM IS ZERO ELSE RETURNS .FALSE.
 4          LOGICAL FUNCTION VSUMZERO(VECT,NUMEL)
 5   C      VECT - ARRAY NAME. NUMEL - NO. OF ELEMENTS IN ARRAY
 6          INTEGER      VECT, VSUM
 7          DIMENSION    VECT(NUMEL)
 8          VSUM = 0
 9          DO 10 I = 1, NUMEL
10             VSUM = VSUM + VECT(I)
11   10     CONTINUE
12          IF ( VSUM .EQ. 0) THEN
13             VSUMZERO = .TRUE.
14          ELSE
15             VSUMZERO = .FALSE.
16          ENDIF
17          RETURN
18          END
```

Figure 3.1 A F77 Program

Note: The numbers at the left extreme end denote line numbers and are not part of the program.

The figure above shows a typical F77 program unit. It is a function subprogram which computes the vector sum of a one dimensional array and returns that sum.

3.3 Symbols and their attributes

The lexical analyser puts the symbols in the symbol table. However, the attributes of these symbols are to be determined at the time of parsing and stored into the symbol table. In F77, the attributes of symbols are not obtained at a single point. They have to be collected by analysing all the specification statements.

Consider, in the Figure 3.1 above, the attributes of symbol VECT. From line 4 (*function declaration stmt*) we gather that it is a formal parameter to the function. Line 6 (*type declaration stmt*) tells that it has type INTEGER. Line 7 (*dimension stmt*) informs that it is a single dimensioned array with the number of elements equal to the value of formal parameter NUMEL. The attribute set of VECT is a collection of all these data. The attribute set of undeclared symbols (e.g. NUMEL above) are to be found from their names by referring to implicit naming type information.

3.4 Tree building

The Treegen tool takes high level input specifications for the structure of different *types* of nodes and the relationship between them and produces definition tables and routines which assist in syntax tree building. The type of a node is uniquely associated with the node name.

There are three different *kinds* of nodes which may be specified.

A *leaf node* is a node which forms a leaf of the syntax tree.
A *list node* has variable number of sons of the same type.
A *other node* has fixed number of sons each of which may be of any type. List nodes and other nodes form the inner nodes of the syntax tree.

A *NULL* node is a special type of leaf node and represents an empty subtree.

A portion of the *NODE* specifications relevant to building the syntax tree of a function subprogram is given below.

```
                        ......................
NODE
function_subprogram  :<
                        son1: opt_comment_stmts ,
                        son2: function_stmt_part,
                        son3: {specification_part, NULL},
                        son4: { execution_part,NULL},
                        son5: end_stmt_part
                       >
                        ......................
```

The tree structure designated by the above specification is shown in Figure 3.2. The line numbers of the statements in Figure 3.1 which would correspond to the different subtrees are also shown.

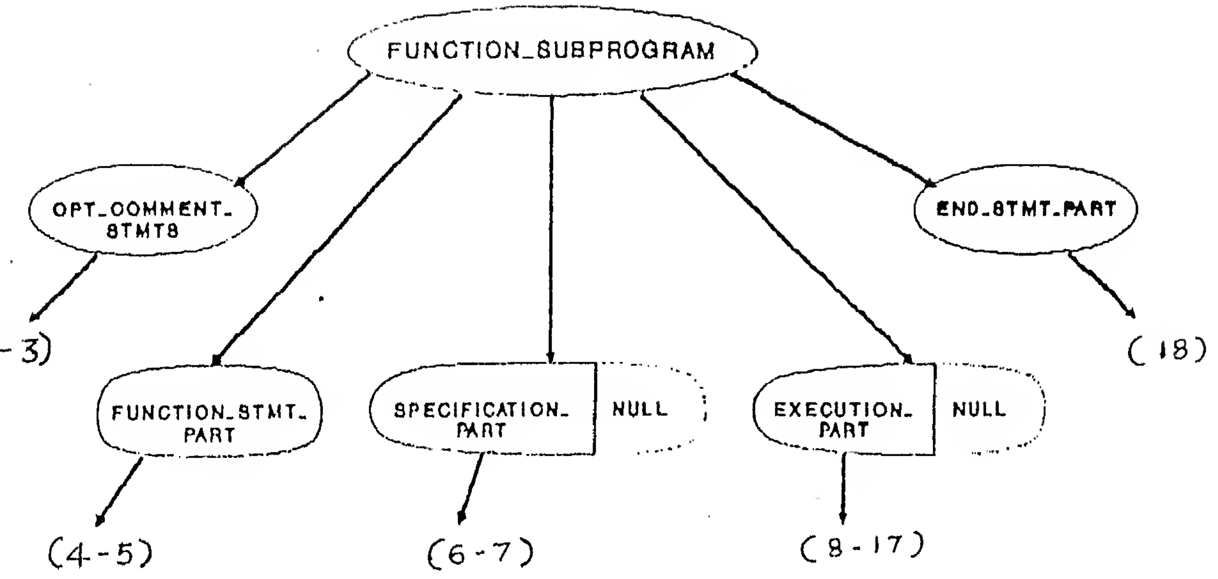

Figure 3.2 Graphical Representation of Sample Node Specification

The structure of the children of the node *function_subprogram* must be specified further in terms of other nodes and they in turn ... etc, till the entire tree is described in terms of *leaf nodes*.

As a second example, the specifications for a *label_do_stmt* would be :

```
                  .......................

    label_do_stmt  :<
                    son1: { LABEL, NULL},     /* leaf */
                    son2: DO,                 /* leaf */
                    son3: INTNUM,             /* leaf */
                    son4: {arith_loop_control,c_for_loop},
                    son5: STEND               /* leaf */
                    >
    arith_loop_control   :<
                            son1: opt_comma,
                            son2: index,
                            son3: / expr /,
                            son4: / expr /,
                            son5: { / expr /, NULL}
                          >
                       .....................
```

Thus the *NODE* specifications describe a template of tree structure into which all possible syntax trees for the language can fit.

The Treegen generates routines which assist in tree building. Some of these are:

1. *NODE makenode(int nodename, int info);*
2. *NODE makeleaf(int nodename, int info, char *uparse);*

Here *nodename* is the user defined name for the node, which will be converted by Treegen into a macro defining an integer. The *info* field would be an integer representing any information the programmer may wish to store in the node. In a leaf node *uparse* represents a character string to be printed in the output while unparsing that leaf.

For other routines the reader is referred to [6].

The tree is built bottom up, starting with leaf nodes and proceeding to the root of the syntax tree. This tree building is done as a part of the actions in the parser as and when the parser recognises different components necessary for building nodes.

A sample of input to Yacc where actions contain call to the tree building routines is shown below.

```
                    .......................

nonlabel_do_stmt   :   _LABEL _DO logical_loop_control _STEND
                        {
                           register int p;

                           p = pop(); /* logical_loop_control  */
                           makeleaf(LABEL,0,$1);
                           makeleaf(DO,0,0);
                           push(p);
                           makeleaf(STEND,0,0);
                           makenode(nonlabel_logical_do_stmt,0);
                           makenode(nonlabel_do_stmt,0);
                        }
                         .......................
```

3.5 Tree Transformation

This is the part of the converter where conversion of F77 syntax tree to C language syntax tree takes place.

The input language to treegen has provisions for specifying RULEs for tree transformations. From these rules for transformations, Treegen generates routines which apply these transformations on the input tree whenever called.

A sample of the RULE specifications for tree transformation is given below.

```
                    .........................

RULE

rule1:      expr_list(son1:VAR,son2:VAR) =>
                      subscript_expr_list(son1,son2).

rule2:      arith_loop_control(son1:VAR,son2:VAR,son3:VAR,
                      son4:VAR, son5:VAR)    =>
                  {
                      if($son5 ==NULL) {
                          makeleaf(INTNUM,0,"1");
                          makenode(primary,0);
                          $son5  =  pop();
                                      }
                  }

                c_for_loop( initialiser(son2,son3),
                   termcheck(son4,son2,son5),step(son2,son5)).
                    .........................
```

What the rules mean:

rule1: rule1 says that the node *expr_list* two of whose sons have labels *son1* and *son2* respectively, should be replaced by a new node of type *subscript_expr_list*. Further, the new node *subscript_expr_list* must have the same two children as those of the old node *expr_list*. In F77 both array reference and function reference are syntactically identical and the grammar for the parser cannot distinguish between them. In C, array and function references have different syntax. To distinguish between the two, in the case of an array the expression list following the array name is transformed into a different type of node called subscript expression list using the above rule.

rule2: The rule2 is more complicated. It has some C code inserted before the target node specification. This code is executed before applying the specified transformation.

In this rule the arithmetic loop control in F77 used for controlling the execution of a DO loop is being converted to a semantically equivalent loop control specification for a *for loop* of C. The *c_for_loop* has three children: the index initialising part - *initialiser,* the loop termination checking condition - *termcheck* and the incrementing step - *step.* These children of *c_for_loop* are functions of the children of the F77 *arith_loop_control* and this information is conveyed by the use of label names *son1 son2* etc..

The routine generated by Treegen which does the ansformation is:

*transform(NODE *ptr_to_rootof_tree);*

ptr_to_root_of_tree is a pointer to the pointer ferring to the root of the tree (or subtree) to be transformed.

The transformer routine traverses the tree in a depth first, left to right order looking for nodes to be transformed as specified by the rules. Whenever a pattern (node listed on the left hand side of a rule) is found it applies the appropriate transformation on it. Transformations on the tree are applied in a bottom up manner. The subtree once traversed is not traversed again normally. However it is possible to specify if needed, that a rematch for a pattern should be searched in a transformed subtree.

3.6 Unparsing the tree

To get the program in the target language (in our case C), the transformed syntax tree has to be unparsed. Along with NODE specifications, one may give specifications for unparsing different nodes. The unparsing specifications contain the following information:

1. Print strings: Strings which must be copied into the output stream of the unparser at different points during the unparsing of the node.

2. User functions: User defined functions which must be executed at various points during unparsing. The node being unparsed is passed as a parameter to these functions. The names of the functions used in this manner must be listed under the FUNCTION section in the specifications.

3. C code: Any C code which must be executed at some point during the unparsing of a node.

4. Which sons and in what order?: During unparsing of a *other node* one may choose to omit one or more sons of the node. That is, the entire subtree following those sons are to be omitted. This information is incorporated by simply omitting the names of those sons in unparse specifications. Also, the unparsing of subtrees under a node may be done in any order by listing the sons in the required order.

The unparse specifications are entirely optional. One may omit these for any node. In such a case that subtree will not be unparsed during the unparsing of the tree.

Some examples of unparsing specifications are given below.

1. Leaf unparsing:

```
                         ..........................
   STEND                        :<              >
                                 <   ";"  doclose_print  >
                         ..........................
```

This specification says that, when the leaf node *STEND* is unparsed, the string ";" should be printed first. Then the unparse string stored in the node *STEND* should be printed. Then a user defined function *doclose_print* should be executed with the node being unparsed as parameter. In this example *doclose_print* checks the*info* field of the node and prints the string "}" as many times as the value of *info*. This is necessary, to see that the non-block do-loops which may be terminated by a labeled action statement are closed correctly.

2. List unparsing:

```
                      ........................
        symname_list      :<
                               [list: SYMNAME]
                          >
                          <"" ", " "" ""  >
                      ........................
```

This unparse specification for the list node *symname_list* has four strings in it. They represent in order, the string to be printed - before unparsing the list node, after unparsing every son of the list, after completing unparsing of all sons and the string to be printed if the list has no sons.

Unparsing of *symname_list* produces a comma separated list of names.

3. Other node unparsing:

```
                      ........................
     main_program    :<
                son1: opt_comment_stmts ,
                son2: { program_stmt_part, NULL},
                son3: { specification_part, NULL},
                son4: { execution_part,NULL},
                son5: end_stmt_part
                     >
                < son1 son2 {gen_other_defs();} son4 son5 >
                      ........................
```

This specification says that while unparsing *ain_program son1* and *son2* must be unparsed after which the user written C code within { and } must be executed. After that *son4* and *son5* should be unparsed in that order. We see that *son3* is omitted in the specification. This means that the son *son3* need not be unparsed.

The function *gen_other_defs()* written as a part of C code, generates definitions for all the variables and functions used in the main program.

3.7 Formatting Unparser output

The output of the unparser represents the target C program in a coded internal representation. To get the text of the C program this must be passed through a formatter written exclusively to be used with the output of the unparser.

Samples of the formatter input and the corresponding output are given below.

Input:

```
                         ...........................
24)24)23)23)22)22)21)21)20)20)19)19)18)18)17)17)12)12)12)
ftoc_7001:12)12)10)10)10)for10)(10)9)9)I = 9)9)9)9)9)9)1;10)
((10)10)10)IMXSIZ) -9)9)I)/10)10)1 > 0  ;10)9)9)I
+=
10)10)1)11)11){11)10)11)11)11)11)10)10)10)DUMMY10)10)[10)
10)10)I][10)10)10)10)10)10)10)3+10)10)10)10)10)10)5] =10)
10)10)10)10)10)011);}12)17)17)17)13)13)if(12)!(12)12)12)
12)S12)12)<12)12)12)(12)12)-12)12)12)12)12)12)5*12)12)12)12)
12)12)6.5)*12)12)12)12)Q(12)12)12)12)12)12)12)5)+12) power
(12)12)12)P,12)12)12)Q)))14){14)13)14)14)14)14)13)13)B=
13)13)13)13)A(13)13)13)13)13)J,
13)13)13)C)14);}14)16)15)15)else16){16)15)16)16)16)16)15)15)B
=15)15)15)15)A(15)15)15)15)15)15)J-15)15)15)15)15)15)1,
15)15)15)D)16);}
                    ........................
```

Output:

```
                    ........................

ftoc_7001   :
     for(I = 1;((IMXSIZ) - I)/1 > 0 ;I += 1){
               DUMMY[I][3+5] = 0;
     }
     if(!(S<(-5*6.5)*Q(5)+power(P,Q))){
               B =  A(J, C);
     } else{
               B    =    A(J-1,    D);
     }
                    ........................
```

3.8 Getting Semantics Right

There are many issues in conversion of F77 to C where care must be taken to preserve the semantics of original program in the translated program. A few of them are discussed here.

1. Arrays : Arrays in F77 have subscripts starting from *1* (not always, but the other ranges are not implemented yet in F2C). In C, array subscripts start with *0*. In translation, the arrays are declared to have one more element than the array in F77 program. The element with subscript *0* is ignored. The other subscripts are mapped one-to-one, i.e., array element *A(30)* of F77 program would be mapped to *A[30]* in the C program.

2. Parameter passing: In F77, parameter passing is by reference always. When expressions or objects with no lvalue are to be passed, their rvalue is stored in a compiler generated temporary variable, and the address of that temporary variable is passed. In C, parameter passing is by value. One can obtain the effect of call by reference by explicitly passing a pointer to the object. In the case of arrays this won't be necessary, because array names are treated as pointers to the beginning of array. In translation, to maintain the F77 semantics, one needs to pass a pointer even for those actual parameters which have no lvalue e.g. an expression, so that the called function can access the value by dereferencing the pointer received through formal parameter. This can be done by, simulating in C, what the F77 compiler does while passing parameters. Assign the expression to a temporary variable generated by the converter. Then take a pointer to that temporary and pass it to the function. This must be done only in the case of objects with no lvalue. Array names may be passed without any change.

3. COMMON objects : Data objects declared in a F77 COMMON statement, will be converted into global data objects in C. All global data object definitions are to be collected in a single file and used as an include file. All the routines of an executable program containing references to these COMMON objects must be translated together so that all the global definitions are in one place.

The translation is done in such a way that an array ence A(expr) for an array which is a part of a common block

BLOCK1 defined in a function subprogram VSUMZERO would be seen by the C compiler which compiles the target program as BLOCK1.VSUMZERO.A[expr].

Apart from the issues discussed above, there are many other issues viz., variables in EQUIVALENCE statements, those in SAVE statements which must be carefully considered while translating F77 to C.

Some sample F77 programs and their translations using F2C are given in the appendix A.

4. TOWARDS A BETTER F2C

4.1 A Complete Converter

Does F2C translate any and every FORTRAN77 program? Unfortunately, the answer is - No.

F2C as it has been implemented now, does not accept the complete language FORTRAN77. Within the time available, the entire language of F77 could not be accommodated in the converter.

To make F2C accept the complete language F77 the following features need to be added.

1. Data object specifications: Some of the data object specifications have not been handled. These include, the COMMON and EQUIVALENCE objects, NAMELIST sets, INTRINSIC and EXTERNAL functions which may passed as parameters, and data initialisation through DATA statements. Also COMPLEX objects haven't been handled. BLOCKDATA subprograms which deal exclusively with specifications have not been taken care of. These features must be added.

2. I/O statements: Many of the I/O handling statements viz, INQUIRE statement, BACKSPACE statement, ENDFILE statement, etc., have not been implemented. Only READ and WRITE statements with limited provisions are implemented. Other file handling features also need to be added.

3. Execution constructs: Computed goto Statement and Assigned goto Statements are not handled now. They must be accommodated.

.2 Testing and Performance Measurement

esting:

A converter needs to be tested in a manner similar to the testing of a compiler. To test a compiler a *test suite* of programs are developed which are designed to test all the features of the source language. This test suite contains both correct programs and incorrect programs. A converter is meant to translate correct source programs only. So to test a converter a test suite of correct programs must be developed. These are to be converted to the target language and the converted programs must be compiled and executed on the target platform to check whether they behave exactly the same way as the programs in the source language by using another test suite of input data sets.

To test F2C a standard test suite was not available. Given the time constraints, developing an entire suite of programs for testing F2C could not have been accomplished. So a different approach was followed. A few programs were written to test carefully as many features as possible. Then a random collection of about fifty F77 programs was made to build a suite of programs to test F2C.

The testing of F2C was done in several stages.

1. **Testing the suite of F77 programs:** The suite of programs was compiled on a HP-9000 Series 850 machine using the HP-F77 compiler. All the programs were accepted by the compiler and no errors were reported by it.

2. **Lexical Analyser:** The lexical analyser was first tested using the few specially written F77 programs and the debugger output was carefully read to see that the lexical analyser behaved as expected on the input. After that the other F77

programs were passed through the lexical analyser and bugs discovered at that stage were removed.

3. Parser: The same approach that was used for the lexical analyser was used here, except that, the input F77 programs now went thorough the lexical analyser developed earlier. The parser was debugged till it behaved as expected. Some simplifications have been made in the grammar, but they do not affect most of the programs.

4. Tree Builder: Similar to the first three phases, the tree builder was tested on the suite of programs and for a few programs the syntax tree built was examined in detail to check correctness.

5. Tree Transformer and Unparser: These two were tested together on a few specially written programs and 'not' on the suite of programs used for testing other stages because the features of F77 mentioned in Section 4.1 above, have been not been implemented for these stages.

Performance Measurement:

To measure the speed of the converter, some large sample F77 programs were made and the the time taken by the converter to translate each of them was determined on a Sun-3 system. A sample of this information is given below:

Program name	No. of lines	CPU time sec User code	CPU time sec System code
prog.f	332	8.1	1.1
init_intmod.f	272	7.8	1.3
transf_md.f	297	8.7	1.7
Total	901	24.6	4.1

It can bee seen that the approximate time for translating about 1000 lines is nearly 30 secs. So the converter speed may be considered to be about 2000 lines of code per minute on a Sun-3 system.

4.3 Sizes of Specifications and Programs

One of the main ideas of this thesis has been to generate programs using language processing tools and reduce traditional programming. The table below shows the sizes of specifications for various tools and the sizes of programs generated by them.

Tool	Specifications no. of lines	C programs generated no. of lines
Lex	325	3346
Yacc	2982	3761
Treegen	1530	3452
Other C routines	494	494
Total	5331	11053

It can be seen that there has been more than 50% saving in the size of the user written code. More than that, there is an enormous amount of time saved because of the ease with which the specifications can be debugged and modified to suit changes in design.

It can bee seen that the approximate time for translating about 1000 lines is nearly 30 secs. So the converter speed may be considered to be about 2000 lines of code per minute on a Sun-3 system.

4.3 Sizes of Specifications and Programs

One of the main ideas of this thesis has been to generate programs using language processing tools and reduce traditional programming. The table below shows the sizes of specifications for various tools and the sizes of programs generated by them.

Tool	Specifications no. of lines	C programs generated no. of lines
Lex	325	3346
Yacc	2982	3761
Treegen	1530	3452
Other C routines	494	494
Total	5331	11053

It can be seen that there has been more than 50% saving in the size of the user written code. More than that, there is an enormous amount of time saved because of the ease with which the specifications can be debugged and modified to suit changes in design.

4.4 From Program to Software Package

A source code converter software package which can be given as a product to end users would need a much larger support than what could be provided by a converter like F2C(even if it were to accept the complete source language).

Any program when moved from one platform to another, is also being moved from one programming environment to another. All the components of the environment used or tacitly assumed by the source program must also be considered while translating it and/or moving it to a different platform. These components include, system calls, library functions, special graphics and CAD support routines, etc.

In such a situation the successful translation and porting of source program would need a series of converters linking source and target environments. One of these converters would translate the system calls, the second would perhaps convert all mathematical library routine calls to those available in the target system, a third would take care of graphics and CAD routines etc., . The source language converter like F2C would indeed be the last one in these series of converters which would finally output programs which would be in the target language and would make use of the program supporting environment available at the target platform.

REFERENCES

[1] *A New Dimension in Software Translation* - An internal report, LEXEME Technology, Pittsburgh, U.S.A.

[2] HP-FORTRAN77 Reference Manual HP-UX, 1988.

[3] Kernighan, B.W. , Ritchie, D.M., *The C Programming Language*, Prentice Hall of India, 1986.

[4] Lesk, M.E., Schmidt, E., *Lex - A Lexical Analyser Generator*, Sun - User's Manual, Programming Utilities and Libraries, 1987.

[5] Johnson, S.C., *Yacc - Yet Another Compiler-Compiler*, Sun - User's Manual, Programming Utilities and Libraries, 1987.

[6] Treegen User's Manual, 1987, T.R.D.D.C., Pune, India.

[7] Aho, A.V., Sethi, R., Ullmann, J.D., *Compilers : Principles, Techniques and Tools*, Addison Wesley, U.S.A., 1986.

[8] FORTRAN - 8X Draft, May 1989, SIGPLAN Special Interest Publication on Fortran.

APPENDIX A

```
C          FUNCTION TO COMPUTE THE VECTOR SUM OF A 1-D ARRAY
C          OF INTEGERS AND FIND IF IT IS ZERO.
C          RETURNS .TRUE. IF VECTOR SUM IS ZERO ELSE RETURNS
C          .FALSE.

           LOGICAL  FUNCTION VSUMZERO(A,NUMEL)

C          A - ARRAY NAME. NUMEL -  NUMBER OF ELEMENTS IN ARRAY.

           INTEGER      A, VSUM

           DIMENSION   A( * )

7001       VSUM = 0

           DO 10  I = 1, NUMEL
              VSUM = VSUM + A(I)
10         CONTINUE

           IF( VSUM .EQ. 0) THEN
              VSUMZERO = .TRUE.
           ELSE
              VSUMZERO = .FALSE.
           ENDIF

           RETURN
           END
```

Sample F1:   A F77 Function Subprogram

```
/*        FUNCTION TO COMPUTE THE VECTOR SUM OF A 1-D ARRAY*/
/*        OF INTEGERS AND FIND IF IT IS ZERO.*/
/*        RETURNS .TRUE. IF VECTOR SUM IS ZERO ELSE RETURNS */
/*        .FALSE.*/

int VSUMZERO(A, NUMEL)
int NUMEL ;
int A [ ];

{
 /*   A - ARRAY NAME. NUMEL -  NUMBER OF ELEMENTS IN ARRAY.*/

 int I ;
 int VSUM ;
 int ftoc_ret_val;

ftoc_7001:
     VSUM = 0;
     for(I = 1;((NUMEL) - I)/1 > 0 ;I += 1){
     VSUM = VSUM+A[I];

ftoc_10:
     ;
 }

 if(VSUM==0){
  ftoc_ret_val = 1;
 } else{
  ftoc_ret_val = 0;
 }
 return ftoc_ret_val;
}
```

Sample C1:   Sample F1 translated into C

112229